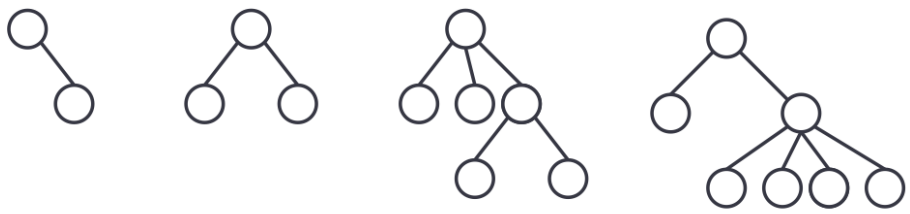
1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery` assumendo implementazione della struttura UnionFind tramite quickFind:

```
algoritmo mystery(n: Int) --> Int
  uf = new UnionFind()
  for i = 1 to n
    uf.makeSet(i)
  endfor
  u = 1; v = n
  while (u <= n)
    uf.union(uf.find(u), uf.find(v))
    u = u*2; v = v-1
  endwhile
  return uf.find(1)
```

Soluzione Innanzitutto, si deve tenere in considerazione il fatto che secondo l'implementazione quickFind, le operazioni sulle strutture UnionFind hanno la seguente complessità: `makeSet` e `find` hanno costo costante $O(1)$, mentre `union` ha costo $O(n)$ nel caso pessimo (con n dimensione della struttura). Veniamo ora all'analisi della complessità $T(n)$ dell'algoritmo. L'algoritmo esegue operazioni di complessità costante ad esclusione dell'operazione `union`. Il primo ciclo esegue una quantità pari ad n di operazioni a costo costante, quindi con complessità risultante $O(n)$, e costruisce una struttura UnionFind di dimensione n. Il secondo ciclo viene eseguito una quantità di volte pari a $\lceil \log n \rceil$. Il corpo del ciclo, nel caso pessimo, ha costo $O(n)$ visto che oltre ad operazioni di costo costante include l'operazione `union` eseguita sulla struttura UnionFind di dimensione n costruita dal primo ciclo. Complessivamente, tale secondo ciclo ha quindi costo $O(n \log n)$. Avremo quindi $T(n) = O(n) + O(n \log n) = O(n \log n)$.

2. Si scriva un algoritmo che prende in input un albero n-ario T e conta quanti sono i livelli che hanno un numero di nodi pari. Nei casi seguenti quindi restituisce rispettivamente 0, 1, 1, 2.



Soluzione L'esercizio si risolve con una visita in ampiezza. Nella coda usata per visitare i nodi si memorizza, oltre al valore del nodo, anche il livello in cui si trova. Ogni volta che si estrae un nodo si verifica se si è raggiunto un nuovo livello. In questo caso si verifica se i nodi del livello precedente erano in numero pari e il conteggio riparte per il nuovo livello. In caso contrario si aumenta il contatore dei nodi al livello corrente. Necessario un controllo finale per verificare se l'ultimo livello ha un numero di nodi pari.

La complessità è quindi $O(n)$.

```
CONTALIVELLIPARI(nodo T) -> int
       livelliConNodiPari = 0;
       Queue q;
       q.enqueue([T,0]);

       livelloCorrente = 0;
       nodiLivelloCorrente = 0;
       while ( q.first != NULL ) do
              p := q.dequeue()

              if (p[1] != livelloCorrente)
                     if (nodiLivelloCorrente % 2 == 0)
                            livelliConNodiPari++
                     nodiLivelloCorrente := 1
                     livelloCorrente := p[1]
              else
                     nodiLivelloCorrente++;
              endif

              foreach x in p[0].children do
                     q.enqueue([x, p[1]+1]);
       endwhile

       if (nodiLivelloCorrente % 2 == 0)
              livelliConNodiPari++

       return livelliConNodiPari
```

3. Progettare un algoritmo che dati due vettori di numeri $A[1..n]$ e $B[1..n]$ restituisce il numero di valori in A che sono presenti anche in B.

Soluzione Una possibile soluzione prevede di ordinare il secondo vettore al fine di poter effettuare ricerche di elementi con costo logaritmico tramite ricerca binaria. Una volta ordinata sarà sufficiente scorrere gli elementi del primo vettore, e contare quanti di questi elementi risultano presenti anche nel secondo vettore (usando appunto una ricerca binaria). Il seguente algoritmo utilizza una variabile ausiliaria `conta` come contatore di quanti valori in `A` sono presenti anche in `B`.

```
algoritmo conta(A[1..n], B[1..n]: Int) --> Int
    // ordina il secondo vettore
    sort(B)
    Int conta = 0
    // per ogni valore nel primo vettore..
    for each x in A
        // ..controlla se e' presente nel secondo
        if (ricerca(x,B,1,n))
            conta = conta+1
        endif
    endfor
    return conta
```

Cont.

```
// algoritmo di ricerca binaria
algoritmo ricerca(x: Int, V[1..n]: Int, s:Int, e:Int) --> Bool
    if (s>e) return False
    else
        Int m=(s+e)/2
        if (x == V[m]) return True
        else
            if (x < V[m]) return ricerca(x,V,s,m-1)
            else ricerca(x,V,m+1,e)
            endif
        endif
    endif
```

dove assumiamo che `sort` sia un algoritmo di ordinamento ottimale (ad esempio heapsort) di complessità $O(n \log n)$. Studiamo la complessità $T(n)$ dell'algoritmo `conta` iniziando l'analisi dall'algoritmo ausiliario di ricerca binaria `ricerca`. Tale algoritmo ha un costo $O(\log n)$. L'algoritmo `conta` esegue prima l'ordinamento di costo $O(n \log n)$. Successivamente esegue un ciclo in cui invoca per n volte `ricerca`, per un costo complessivo $O(n \log n)$. Complessivamente avremo $T(n) = 2 \times O(n \log n) = O(n \log n)$.

4. Bisogna verificare se dato un budget B è possibile costruire una rete stradale che permette di collegare un insieme di punti su una mappa. Solo fra alcune coppie di punti è possibile costruire una strada, e ogni strada di collegamento fra due punti ha un suo costo. Si assuma che tali possibili strade ed i relativi costi siano rappresentati tramite un grafo non orientato pesato $G = (V, E, w)$, ovvero V è l'insieme dei punti sulla mappa, $(u, v) \in E$ se e solo se è la possibile costruire una strada da u a v, e $w(u, v)$ è il costo di costruzione di tale strada. Progettare un algoritmo che dato il grafo non orientato pesato G ed il budget B restituisce un booleano; *true* se è possibile costruire una rete stradale, con un costo complessivo minore o uguale a B, che permette da un qualsiasi punto di raggiungere un qualsiasi altro punto (eventualmente attraversando altri punti intermedi), *false* altrimenti.

Soluzione Assumendo che esista la possibilità di costruire tale rete stradale, ovvero il corrispondente grafo è connesso, collegare tutti punti sulla mappa coincide con definire un sottografo che connette tutti i vertici del grafo corrispondente. Un tale sottografo di costo complessivo minimo corrisponde con il cosiddetto minimum spanning tree del grafo. Il costo di tale albero puè essere calcolato, ad esempio, tramite l'algoritmo di Kruskal. Al termine dell'esecuzione dell'algoritmo si verifica semplicemente se il costo ottenuto è inferiore oppure no al budget disponibile.

Il seguente algoritmo è una versione di Kruskal che non calcola l'albero di copertura ma solo il suo costo complessivo utilizzando una variabile di appoggio `costo`.

```
algoritmo mappa(Graph(V,E,w): G, Number: B) --> Bool

    // inizializzazione strutture dati
    UnionFind UF
    for i = 1 .. G.numNodi()
        UF.makeSet(i)
    endfor
    Number costo = 0

    // ordina gli archi di E per peso w crescente
    sort(E, w)

    // controlla gli archi appartenenti al minimum spanning tree
    for each {u,v} in E do
        Tu = UF.find(u)
        Tv = UF.find(v)
        if (not (Tu == Tv)) then // evita i cicli
            costo = cost + w(u,v)
            UF.merge(Tu, Tv)   // unisci componenti
```

Cont.

```
    endif
endfor

// controlla se il costo del MSP e' inferiore al budget
return (costo <= B)
```

La complessità di tale algoritmo corrisponde con la complessità dell'algoritmo di Kruskal, ovvero $O(m \log n)$ con $m = |\mathtt{E}|$ e $n = |\mathtt{V}|$.

The End.